Exhaustive testing of safety critical Java

Kalibera, Tomas; Parizek, Pavel; Malohlava, Michal; Schoeberl, Martin

Published in:
Proceedings of the 8th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2010)

Link to article, DOI:
10.1145/1850771.1850794

Publication date:
2010

Document Version
Early version, also known as pre-print

Exhaustive Testing of Safety Critical Java

Tomas Kalibera, Pavel Parizek, Michal Malohlava
Department of Distributed and Dependable Systems
Charles University
{kalibera,parizek,malohlava}@d3s.mff.cuni.cz

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

With traditional testing, the test case has no control over non-deterministic scheduling decisions, and thus errors dependent on scheduling are only found by pure chance. Java Path Finder (JPF) is a specialized Java virtual machine that can systematically explore execution paths for all possible schedulings, and thus catch these errors. Unfortunately, execution-based model checkers, including JPF, cannot be easily adapted to support real-time programs.

We propose a scheduling algorithm for JPF which allows testing of Safety Critical Java (SCJ) applications with periodic event handlers at SCJ levels 0 and 1 (without aperiodic event handlers). The algorithm requires that deadlines are not missed and that there is an execution time model that can give best- and worst-case execution time estimates for a given program path and specific program inputs.

Our implementation, named RSJ, allows to search for scheduling dependent memory access errors, certain invalid argument errors, priority ceiling emulation protocol violations, and failed assertions in application code in SCJ programs for levels 0 and 1. It uses the execution time model of the Java Optimized Processor (JOP). We test our tool with Collision Detector and PapaBench application benchmarks. We provide an SCJ version of the C PapaBench benchmark, which implements an autopilot that has flown real UAVs.

Categories and Subject Descriptors

C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.2.4 [**Software Engineering**]: Software/Program verification—*Model checking*

General Terms

Verification, Algorithms, Experimentation

JTRES'10 August 19–21, 2010 Prague, Czech Republic

Keywords

SCJ, Java PathFinder, model checking, real-time Java

1. INTRODUCTION

There is a growing need for automated and semi-automated verification and testing tools for the purpose of certification. While existing systems are being certified solely based on software engineering processes and manual verification (e.g., certification for aviation – DO-178B [20] and [7]), the purely manual approach does not scale with the increasing amount of safety-critical code and the increasing hardware complexity.

A particular class of verification tools are those built on top of execution-based model checkers, which both execute programs forward as well as backtrack, while systematically exploring the program state space [29, 17, 8, 16]. The model checkers have control over otherwise non-deterministic aspects such as scheduling decisions, allowing to explore also corner cases. While the model checkers implement a plenty of optimizations of the state space traversal, they often fail to explore all possible states due to state explosion: the number of program states is usually exponential of the program size, and thus the checking process runs out of time or memory. However, even if exploring the whole state space is not practically possible, model checkers can be used as bug-hunting tools that can systematically test program executions under different schedulings, which cannot be done by normal tests. Also, bug-hunting tools need not be certified by themselves.

Execution-based model checking of real-time Java programs, be it for Real-time Specification for Java (RTSJ) [4] or Safety Critical Java (SCJ) [10], brings significant challenges compared to model-checking of standard Java. The scheduler in the model checker has to be replaced by one that implements fixed priority preemptive scheduling with no time-slicing and with support for periodic activities. The key problem is that the model checker needs to have a certain notion of the passage of time, for which it has to know execution time bounds of the code it executes. Although scheduling decisions are made using a known and deterministic algorithm in real-time systems, they depend on the wall-clock time and on execution time, which are not perfectly deterministic: they can be subjected by temperature, the precision of the system clock, and initial conditions such as placement in memory and its influence on cache misses or on the timer phase. The scheduling decisions are thus

still non-deterministic to some degree and this level of non-determinism is magnified in a model checker, which necessarily works with a (timing) model of the hardware that cannot be perfectly precise.

Interestingly, a model checker for plain Java does not need to have any notion of the passage of time to explore all possible interleavings: with time-slicing, native concurrency on multi-processors, and without strict priority enforcement, any runnable thread can be scheduled at any time.

In our earlier work [19], we proposed a scheduling algorithm for Java Path Finder that is based on a notion of time that uses the knowledge of periods and start times of periodic tasks, under the assumption that no deadlines were missed. The model did not allow to actually check that deadlines were not missed, leaving this task to other tools. In this work, we propose a different algorithm, which relies on bytecode level execution time model for a particular processor. Both algorithms explore a superset of schedulings that are possible on a real system. This algorithm is much more efficient, as it explores fewer schedulings of those that are not possible. The algorithm we propose in this paper allows to actually check that deadlines are not missed (again given the specific program inputs). The JPF extension described in this paper supports the SCJ API, while our earlier work supported the RTSJ API. We provide an empirical comparison of the two approaches.

To evaluate our algorithm we implement RSJ, an extension of Java Path Finder (JPF) [29, 1]. RSJ supports a subset of SCJ levels 0 and 1 (most importantly it does not support aperiodic event handlers) and uses a timing model of a hardware Java implementation, the Java Optimized Processor (JOP) [24]. The tool can discover memory access errors, race conditions, priority ceiling emulation protocol violations, and some other run-time errors that originate from the application logic, such as dereferencing of a null pointer, invalid arguments to certain library calls, array bounds violation, division by zero, or failed assertions in program code. The tool can also report execution time bounds for individual releases of periodic event handlers when executed on JOP, but still only for a particular set of inputs given to the tool. Therefore, RSJ *is not* a tool for worst-case execution time (WCET) analysis. It is, instead, a bug-hunting tool for real-time Java programs.

We run RSJ with two application benchmarks, Collision Detector [12] and PapaBench [18]. For this, we have translated PapaBench from C to Java. We validate some aspects of the tool also with the Kfl, Lift, and UdpIp [25] benchmarks.

In summary, our contribution is:

- A scheduling algorithm for an execution-based model checker for SCJ that explores all possible executions of a program on a given platform, particularly supporting most of the SCJ real-time scheduling features.
- RSJ, a prototype of a testing tool for SCJ that validates the algorithm. It systematically explores different schedulings of a program when testing, assuming the JOP processor as the target platform. A subset of SCJ is supported. RSJ can detect memory access errors, race conditions, and a range of other errors.
- SCJ and RTSJ versions of PapaBench, an open-source real-time benchmark built of an autopilot of a UAV. The original C benchmark code has flown a real UAV.

We provide sources of RSJ with benchmarks and scripts we used for evaluation in this paper.[1] We also merged the functionality of RSJ into RTEmbed, a JPF extension for checking real-time and embedded programs, which can be downloaded from the official JPF website.[2]

2. BACKGROUND

2.1 Java Path Finder

As the core contribution of our work is a scheduling algorithm for an execution-based model checker, we start with a rather broad view on model checking and with characterization of Java Path Finder, a model checker we use. The understanding of Java Path Finder, described below, is later needed in the description of our scheduling algorithm.

Software model checking is a technique for finding properties about execution of programs based on algorithmic analysis of the programs. Execution-based model checkers analyze programs by executing them on a modified runtime system, i.e. Java virtual machine. These runtime systems allow saving a program state and backtracking to a previously saved program state. The model checkers aim to explore all program states that can be reached with any (non-deterministic) inputs and any (non-deterministic) scheduling of concurrent threads. Typically, only these two sources of non-determinism are assumed. Common properties to check are safety properties, which assert that certain error states of a program are not reached, e.g., a state in which a memory assignment error occurs. Model checking is an expensive process due to the state explosion problem. However, many optimizations and heuristics for different tasks of model checking have been studied and implemented [11]. Currently, several bug-hunting tools are available and have been used successfully to discover subtle errors in network protocols, concurrent phone-switching software, or file systems [8, 16]. The practical advantage of bug-hunting tools is that they do not need to traverse the whole program state space to be useful, as long as they can discover error states.

Java Path Finder (JPF) [29, 1] implements an interpreting Java virtual machine that allows to save the visited program states and restore them as needed. JPF can detect already visited states (*state matching*), so they are not unnecessarily explored multiple times . JPF backtracks to program states called *choice points*, which can be either data choices (reading an input value) or thread choices (the scheduler can decide to run another thread). We assume the usual practice in testing where data are fixed inputs of the test, yet JPF also has support for symbolic execution and abstraction that allows to identify sets of inputs that lead to different program states.

In standard Java running on a system with native scheduling, any runnable thread can preempt the current thread at any time. However, it is unnecessary to explore all possible interleavings of threads as long as the threads are running code that has no visible effects to other threads – JPF thus creates thread choice points only before accesses to shared fields, synchronization, or other scheduling-relevant instructions (*partial order reduction*). JPF is highly customizable, so that one can further reduce the amount of options in

[1] http://d3s.mff.cuni.cz/~kalibera/scjcheck

[2] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/rtembed

thread choice points as needed for a particular verification purpose, as well as direct the search by heuristics.

JPF is written in Java and runs on top of a host Java virtual machine. To speed-up the checking process when only a part of a large application needs to be checked, one can instruct JPF to run certain components in the host machine. All native functions have to be handled specially and all operations that have influence outside the VM have to be handled such that backtracking is possible. JPF does not keep track of wall-clock time as it would appear to the running application on a real VM. The assumption is that typical applications do not base their functionality on wall-clock time, which works well for non-realtime applications and applications intended to be portable across systems. Any timed sleep invoked by the application has the same effect: it creates a thread choice point, so any runnable thread including the one that invoked the sleep operation can continue executing. This solution may lead to exploration of impossible execution paths, but it will always cover all execution paths that are possible. This handling of sleeps, however, becomes incorrect when we introduce strict priority enforcement of threads, which is the fundamental property of real-time systems. Once we do this, the top priority thread will starve out all other threads no matter how long it is sleeping (i.e. waiting for the next event to handle).

2.2 The Java Optimized Processor

Java processor JOP [24] is an implementation of the JVM in hardware. The bytecodes of the JVM are the native instructions of the processor. JOP has been optimized to be time-predictable, while still performing well. The execution pipeline is a 4 stage in-order pipeline. The execution time of individual bytecodes are independent of each others. These properties enabled building worst-case execution time (WCET) analysis tools for Java that target JOP [26, 9]. JOP is supported as one possible target for R_{SJ}. JOP is selected as the first target, as it is a time-predictable JVM where the timing model is available. The execution time of individual bytecodes, the instruction set of the JVM, is known cycle accurate. Most bytecodes have a constant execution, which means that the WCET and the best-case execution time (BCET) are equal. Variability in the execution time mainly results from the instruction cache. To simplify WCET analysis of the instruction cache, JOP caches whole methods [21]. Therefore, cache misses can only happen on method invocations and on a return from a method. This method cache also simplifies the timing model for R_{SJ}. We have included the possible hits or misses into the method cache in the timing model as BCET and WCET for the invoke and return bytecodes.

As a simplification we have not modeled the cost of preemption, scheduling, and the context switch. A safe assumption for the preemption cost would be the invalidation of the cache. Therefore, on a context switch the method of the newly executable thread needs to be loaded into the method cache. Furthermore, the stack cache of the preempted thread needs to be saved and the stack cache content of new thread needs to be restored. As the stack size is bounded, this execution time is bounded. The worst-case execution time of the scheduling decision is linear with the number of schedulables. For SCJ application the number of schedulables is fixed and therefore the execution time is bounded.

Note that the timing model for R_{SJ} is a much simpler task than a full-fledged WCET analysis. As R_{SJ} has the exact knowledge of program inputs, program code, program state, and execution history, we could have calculated exact execution time using a cycle-accurate JOP simulator. However, thanks to the fact that the scheduling algorithm of R_{SJ} accepts also intervals for execution time, the execution time model can be simpler than cycle-accurate simulation. Another possible use of imprecise execution time model for R_{SJ} would be support for platforms that do not have a fully deterministic execution time.

2.3 Scheduling in SCJ

Safety Critical Java (JSR-302, [10]) is an upcoming technology based on the RTSJ, which should allow certification. While the draft is still being subjected to frequent changes, the core aspects of scheduling and memory management, which we summarize below, are reasonably stable. They allow us to formulate the scheduling algorithm for model checking and to describe memory assignment rules which need to be checked. SCJ distinguishes three levels of complexity, numbered 0, 1, and 2. We only focus on levels 0 and 1 in this work. Although SCJ allows multi-processors at level 1, we restrict our analysis to uniprocessors, as certification of multi-processor Java applications seems not to be possible in the near future. We also restrict the supported features of SCJ whenever they would lead to state explosion or overly complex implementation.

An SCJ application is composed of *missions*. Each mission has *schedulable objects* (a.k.a. *schedulables*) and *mission memory* shared by these schedulable objects. Each schedulable also has its own *private memory*. The system then has *immortal memory*. Objects in immortal memory can only have pointers to immortal memory. Objects in mission memory can have pointers to immortal or mission memory of the same mission, and objects in private memory can have pointers to immortal memory, mission memory of its mission, and the same private memory. Private memory areas can be nested, following the same rules for pointers between areas.

Each mission is first initialized by a single thread, in which all mission memory and immortal memory allocations of the mission take place. This thread also creates schedulable objects, which can be either periodic (*periodic event handlers*) or aperiodic (*aperiodic event handlers*). Aperiodic event handlers are released at times unknown to the application, typically originated by external hardware, and we ignore them here – supporting them in this unconstrained form would lead to state space explosion. Periodic event handlers are given a priority, period, and start time. The start time is absolute or relative to the time when initialization is over. Once initialization is over, the schedulables can be released.

We do not support absolute release times. If an application needs to release periodic handlers with fixed relative offsets, it would be natural to have these handlers in the same mission, which in turn does not require support for absolute start time. We discuss implications of potential support for aperiodic handlers and absolute time in more detail later, in the context of the algorithm we propose.

On every release of a schedulable, its private memory is initialized and entered. The memory is implicitly destroyed when the release completes. Mission memory is destroyed

when the mission terminates. Immortal memory is only destroyed when the VM is shut down.

At *level 0*, all schedulables are run by a single infrastructure thread; there is no preemption and no concurrency. There is only one mission at a time. Only periodic event handlers are supported. The schedule is static and cyclic. Each cycle is formed by the same sequence of frames, each having a duration and a sequence of handlers to run. While this schedule is expected to be specified according to priorities, periods, and start times of the handlers, the SCJ runtime ignores these – it only uses the static schedule. At level 0, for a given set of inputs, a model checker that explores execution paths along all possible schedulings thus cannot discover more errors than actual execution of the program – there is only one scheduling. RSJ can still calculate worst case execution time bounds *for the given program inputs*. As of now, level 0 support is beneficial mostly for debugging of the tool and of applications that can run both at levels 0 and 1. RSJ could, however, be extended to support checks that regular tests cannot do, such as systematic exploring of different execution paths given by program inputs.

At *level 1*, every schedulable is run by its own infrastructure thread, and potentially concurrently with other schedulables. RSJ can discover more errors than testing, because it explores potentially all valid interleavings of the schedulables, and it may be that some errors are only present in a particular interleaving. RSJ can also detect race conditions. At this level, there is still a single mission at a time. It means, in particular, that all schedulables are started relative to the same time origin. This level also supports aperiodic event handlers. Java monitors are allowed and use priority ceiling emulation protocol. Calls to `Object.notify` and `Object.wait` are not allowed at this level.

3. SCHEDULING ALGORITHM

We propose the following scheduling algorithm for a model checker that executes SCJ programs. Note that the algorithm is more complex than one of a real VM, because it has to cover all possible execution paths (not just one) and it has to work with an interval of the actual wall-clock time $t = [t_{min}, t_{max}]$. Our algorithm bases the interval on the estimated execution time along the explored execution path. t is undefined from the program start up to the first release of a periodic event handler.

The application can indeed use a call to `Clock.getTime` to get the current wall-clock time. We always return a lower bound, t_{min}, or zero with a warning when t_{min} is still undefined. While the lower bound may not work well for every possible use of time in the program, it is easy to trace these operations and it is unlikely they will be heavily present, provided that SCJ can start handlers relative to the common base – end of initialization of a mission.

3.1 SCJ Level 0

The scheduler proceeds as follows. t_s is the wall-clock time of frame start, which is then also the time when the first handler of the frame is released. d is the duration of the current frame.

1. After all initialization code has finished or blocked, $t := [0, 0]$, $t_s := 0$. The first handler from the frame is "released" (its `handleAsyncEvent` method is called).
2. After `handleAsyncEvent` finishes, t_{min} is increased by the best-case execution time estimate and t_{max} by the worst-case execution time estimate of the executed `handleAsyncEvent`. If $t_{max} > d$, the possibility of frame overrun is reported. Assuming frame overrun in fact did not happen, t_{max} can be set to d and the checking can continue. This step is repeated until there are no more handlers in the current frame.
3. Once the last handler of the current frame has finished executing, we update the time of frame start for the next frame: $t_s := t_s + d$. We advance the current time to the start time of the next frame: $t := [t_s, t_s]$, and we continue by executing handlers from the next frame (Step 2).

3.2 SCJ Level 1

At level 1, there is only one mission at a time, thus only a single time base starting with the first release of a periodic event handler at time $[0, 0]$. For each periodic event handler $i = 1, .., n$, we know release times of the handler: $t = [o_i + j \cdot T_i, o_i + j \cdot T_i]$, where o_i is the relative start time (*offset*), T_i is the period, and $j \geq 0$ is the release index.

At time $t = [0, 0]$, we know about all periodic event handlers, and we could create an (infinite) ordered queue of release events $Q = e_1, e_2, e_3, \ldots$, such that $t^{e_1} < t^{e_2} < t^{e_3} < \ldots$ holds for release times $t^{e\bullet}$. Each event corresponds to the release of one or more periodic event handlers ($H^{e\bullet}$). The algorithm, however, only needs to query about the first element of the queue and eventually to remove this element. It is thus sufficient to generate the queue on the fly, which is possible as follows. We remember the index of the last release of every periodic handler that has been added to the queue. Whenever we remove an element from the queue, we calculate the next release time of every handler. Out of these times, we take the minimum as the time of the new release event. The set of handlers of this event would be all handlers that should be released at this minimum time. Let $p^{e_1}, p^{e_2}, \ldots$ be the highest priorities of event handlers in sets $H^{e_1}, H^{e_2}, \ldots$

The algorithm described below is also pseudo-coded in Figure 1. Line numbers in the figure refer the steps below:

1. There is no runnable handler at this time and the system is idle. We take the first element e_1 from Q, $t := [t^{e_1}, t^{e_1}]$, make all the handlers from H^{e_1} runnable, and we start executing the first top priority handler. We remove e_1 from Q.
2. After executing a bytecode instruction, we update current time by adding the best-case execution time estimate of the executed instruction to t_{min} and the worst-case estimate to t_{max}. Note that the executed instruction may have lead to priority changes or may have blocked or suspended the handler.
3. If $t^{e_1} \leq t_{min}$ (the release event has to happen now or it should even have happened earlier), we unconditionally remove this event from the queue and release all its handlers. We keep doing this until we reach a state where $t^{e_1} > t_{min}$.
   Note that our VM implicitly increments current time with the resolution that depends on execution time estimates of the executed instructions. The maximum possible increment, which is the maximum estimated execution time of any instruction, thus becomes the timer resolution. We could execute larger code blocks

```
run() {
6  while(hasNextMission()) {
6    initialize()
6    doWhenIdle()
5    while(inMission()) {
5      executeInstruction(nextInstruction())
5      if (idle()) {
5        doWhenIdle()
5      }
5    }
6  }
}

doWhenIdle() {
1  executeEarliestEvent()
1  enterScheduler()
}

executeInstruction(insn) {
2  [i_bcet, i_wcet] = estimateTime(programState, insn)
2  t_min = t_min + i_bcet
2  t_max = t_max + i_wcet
3  while (t^e1 ≤ t_min) {
3    executeEarliestEvent()
3  }
4  if (t^e1 ≤ t_max) {
4    p_max = maxRunnableHandlerPriority()
4    if (p^e1 > p_max) {
4      if (nondetChoice()) {
4        executeEarliestEvent()
4      }
4    }
4  }
}

executeEarliestEvent() {
  t = [t^e1, t^e1]
  setRunnable(H^e1)
  Q.removeHead()
  Q.addNextEvent()
}
```

Figure 1: Scheduling algorithm for level 1.

than single instructions, but for the price of a worse timer resolution and consequently higher release jitter. Note that it is possible that an event should have happened earlier due to two reasons: (a) we missed it because of limited timer resolution, (b) we intentionally "missed" it as an optimization because releasing it earlier would not have had any impact on the program state (detailed in the next step).

4. If $t^{e_1} \leq t_{max}$ (the next release event may happen now as well as later, but we do not know for sure, because we do not know the exact current time), we may have to make a non-deterministic choice as if to execute release e_1 or not. Note that we know that the event can already happen because of the previous step which guaranteed $t^{e_1} > t_{min}$. We want to avoid doing a non-deterministic choice whenever possible, because it is costly. We can avoid doing it whenever there already was a runnable handler in the system with a higher priority than those handlers that would be released. So if the maximum priority of all runnable handlers is smaller than p^{e_1}, we make a *non-deterministic choice* as if to execute the release event e_1 or not. If we execute the event, we also update the current time $t := [t^{e_1}, t^{e_1}]$.
5. If the system became idle, we continue at Step 1. Otherwise, the scheduler will continue executing the runnable handler with the highest priority (Step 2). Although we could avoid entering the scheduler if we (a) did not release any handlers and (b) the handler that executed last will not be descheduled, such an optimization would unlikely be beneficial – the scheduler decision is one that is easy to make.
6. Eventually, a mission may terminate and another/the same mission may be started. If this happens, the data structures are re-set for the new mission and the algorithm continues in Step 1.

Supporting aperiodic event handlers would lead to state explosion at level 1. Aperiodic event handlers do not have a minimum inter-arrival time. They still have deadlines, but even assuming deadlines were met would not speed-up the checking process, as there is no minimum inter-arrival time. Technically, we would thus have to make a non-deterministic choice at every single instruction as whether to release a given not-released aperiodic event handler. In particular, we would have to explore also paths where the aperiodic handlers would be starving out all handlers of lower priority. We could again restrict to releasing only aperiodic event handlers of higher priority than the highest priority event handler in the system (optimization like in Step 4). We would have to introduce special handling for idle time: we would not be able to simply "skip" the idle time like in the case with periodic event handlers only (Step 1), because we would not know how long this time should be; a possible implementation would be to, at a pre-specified rate, keep advancing the current time and non-deterministically deciding which aperiodic event handlers to release.

Note that while the real-time scheduling theory provides means to implement aperiodic events using periodic events, i.e. constant-bandwidth servers or a polling server, we cannot use these in RSJ without modifying the scheduler behavior dictated by the SCJ specification. Still, an application programmer could modify an application of interest to use periodic event handlers to model aperiodic events, and then check such application with the present version of RSJ.

Supporting absolute time. If absolute time was used for starts of periodic event handlers, we would have to keep track of two time bases (start of application and start of mission execution phase) as opposed to only one (start of mission execution phase). We would thus essentially get an extra mission, formed of the handlers started using absolute time. We would need a special queue for the release events of these handlers and the algorithm would be more complex.

To support the notion of absolute time we would have to estimate the execution time of initialization, so that we could relate the two time bases. We would also require the user to specify absolute time at program start, most likely it would be zero.

Supporting SCJ level 2. Similarly to absolute time for starts of periodic event handlers, level 2 would bring up multiple time bases, one for each mission. Each mission would have its own queue of release events and the scheduling algorithm would have to correctly select events from multiple queues such that it did not violate the known bounds on times of the next release event from each mission/queue.

4. EVALUATION

We implemented the proposed algorithm in $\mathsf{R_{SJ}}$, an extension of JPF. We only use the JPF API designed for JPF extensions, without modifying of the JPF core. This choice should reduce the maintenance costs of porting to future JPF versions, but it complicates the implementation and makes certain optimizations impossible, such as avoiding creation of a new state at a deterministic thread switch. We support a subset of SCJ that covers all aspects of the specified algorithm and that allows to execute our tests and two application benchmarks, Collision Detector and PapaBench. We use the version of SCJ specification as of May 2010.

The goals of our evaluation are:

- To demonstrate that $\mathsf{R_{SJ}}$ can find errors as intended: show that it does not find spurious errors in tested code that we believe is correct and it finds manually injected errors (Section 4.3).
- To demonstrate that the timing model implemented in $\mathsf{R_{SJ}}$ corresponds to the hardware implementation of JOP (Section 4.4).
- To compare performance of exhaustive testing using the proposed algorithm that uses the JOP timing model ($\mathsf{R_{SJ}}$ tool) against our earlier algorithm that is timing-model independent ($\mathsf{R_J}$ tool) (Section 4.5).

The source code of $\mathsf{R_{SJ}}$, the benchmarks, tests, and scripts we used is available from http://d3s.mff.cuni.cz/~kalibera/scjcheck. We have executed the experiments on a Linux machine with a 64-bit 3GHz Intel Xeon CPUs, 4MB L2 cache and 32GB RAM (JPF only uses a single core).

4.1 Collision Detector Benchmark

Collision Detector (CDx) is a real-time application benchmark [12] that simulates an aircraft collision detector. It is available in multiple variants and with a configurable workload. It contains a hard real-time *detector thread* that periodically receives a radar frame with the current positions of aircraft. Based on the current and the previous frame the detector thread detects whether any two aircraft are on a collision course. The benchmark code has not been deployed on a real system. We use this benchmark in two versions, `nosim` and `sim`.

In the `nosim` version, which is similar to the original SCJ version of CDx [12], the radar frames are being generated synchronously by the detector thread, providing a simple synthetic air traffic. Thus, there is only one active thread throughout the execution of the benchmark. We also ported the `nosim` version to RTSJ, so that we can compare $\mathsf{R_{SJ}}$ and $\mathsf{R_J}$ ($\mathsf{R_J}$ is for RTSJ programs). We modified the original version of CDx [12] to conform to the current version of SCJ and to support level 1 in addition to level 0.

In the `sim` version, which is based on the full RTSJ version of CDx, the radar frames are being generated concurrently by a *simulator thread*, using a configurable air traffic simulator that is far more complex than the hardcoded air traffic in the `nosim` version. The simulator thread is storing the generated frames for the detector thread in a wait-free queue implemented as a circular buffer. We have modified the original version of CDx to only use scoped and immortal memory (the original one uses heap for the simulator thread), and we again created both a RTSJ and a mirror SCJ version, so that we can compare $\mathsf{R_{SJ}}$ and $\mathsf{R_J}$.

We injected errors into the two benchmarks, as proposed in [13]. Some of the errors are based on real errors that were discovered during the benchmark development, some are purely synthetic. SCJ versions of `nosim` and `sim` can be run both at level 0 and level 1. The versions we use are summarized in Table 1.

Name	RT API	Simulator
`cdx-l0-nosim`	SCJ L0	No
`cdx-l1-nosim`	SCJ L1	No
`cdx-rtsj-nosim`	RTSJ	No
`cdx-l0-sim`	SCJ L0	Yes
`cdx-l1-sim`	SCJ L1	Yes
`cdx-rtsj-sim`	RTSJ	Yes

Table 1: Versions of the CDx benchmark we use.

	Task	Period	Depends on
Autopilot			
A1	Radio control	25ms	F2
A2	Stabilization	50ms	A1, A4, A7, S3
A3	Fly-by-Wire link	50ms	A2
A4	Reporting	100ms	
A5	Navigation	250ms	S2
A6	Altitude control	250ms	A5
A7	Climb control	250ms	A6
Fly-by-Wire			
F1	Test PPM	25ms	
F2	Send data to autopilot	25ms	F1
F3	Check fail-safe	50ms	
F4	Check autopilot	50ms	A3
Simulator			
S1	Environment simulator	25ms	
S2	GPS interrupt	50ms	
S3	Infra-red interrupt	250ms	

Table 2: PapaBench tasks and their properties.

4.2 PapaBench Benchmark

The PapaBench benchmark [18] is based on the Paparazzi project [2], which is an open-source implementation of a UAV. The software and hardware provided by this project has successfully flown several UAVs. The software part of the project includes hard real-time on-board software and non-realtime support software that runs on the ground station. The ground station receives telemetry data from the aircraft and allows to trace the flight in a GUI. The on-board software includes a fly-by-wire unit and an autopilot. The fly-by-wire unit receives radio commands and controls the servos. The autopilot unit receives data from sensors (GPS and infra-red) and sends commands to the fly-by-wire unit. The autopilot is fully autonomous and follows a flight plan written in a high-level language. The intended payload is a video camera which can communicate to the ground in real-time. For emergency situations and testing, the plane can also be controlled remotely from the ground, overriding the autopilot. The project also includes a simulator of the hardware devices present in the plane and a physical model of the plane, so that it is possible to debug the autopilot and/or the ground station software off-line.

The on-board hard real-time software has been extracted into the PapaBench benchmark [18], which can be run on a desktop machine. The benchmark is merely a four-year-old snapshot of the Paparazzi on-board code, thus it is simpler than today's Paparazzi code base, but it still includes real code that has flown. It consists of 13 periodic tasks that

follow a cyclic schedule and of 6 interrupt sources (sensors, servos, communication bus). The original PapaBench was not intended for functional execution (processing real data from hardware), but rather for evaluation of WCET analysis tools. It can, however, be connected to the Paparazzi simulator of the on-board hardware with minor effort.

We have rewritten the C code of PapaBench into Java (SCJ levels 0 and 1, RTSJ) and added the necessary parts of the Paparazzi simulator for a functional execution. Particularly, we have included a physical model of the plane sensors, a corresponding physical model of the plane based on Paparazzi code, and suitable flight plans. The simulator is also written in Java and the interrupts are modeled by periodic tasks, so that the benchmark can be run on any SCJ implementation.

The original Paparazzi on-board code is deployed on two physical units communicating via an SPI bus. In the Java implementation, the whole functionality is merged into one application and deployed and executed in a single VM together with the simulator. However, a hardware abstraction layer for the SPI bus, sensors, and servos was preserved in the Java code. The current Java implementation does not simulate commands sent from the ground station – the simulation only focuses on an autonomous flight.

The resulting Java implementation includes 14 tasks, out of which 7 tasks control the autopilot unit and handle navigation, airplane control, and communication with the fly-by-wire unit. The fly-by-wire unit contains 4 tasks that receive commands from the ground (they actually never receive any command in the current implementation, as we do not simulate commands sent from the ground), send and receive commands from the autopilot unit, and configure the servos. The remaining 3 tasks are dedicated to the environment simulator, GPS interrupt, and infra-red device interrupt. The frequency of the tasks varies from 4Hz to 40Hz as specified in the PapaBench benchmark. The properties of tasks, including their dependencies, are summarized in Table 2. The original PapaBench implementation is lock-free with cyclic scheduling, which is reflected in the Java code.

In addition to the flight plans we created for RSJ experiments, we provide a longer one suitable for testing native VMs. We are also working on support of the data connection link from the plane to the ground, so that the ground station software of Paparazzi could be connected to the benchmark, allowing to visualize the actual flight.

4.3 Validation of Error Discovery

We use three sets of patches that inject errors to our benchmarks. One set is for Collision Detector `nosim`, one for `sim`, and one for PapaBench. Each error patch can be applied to both level 0 and level 1 code. The errors are created manually, some of them are real errors we actually found in earlier versions of the Collision Detector benchmark.

The patches for the Collision Detector `nosim` include an array out of bounds exception (invalid upper bound in a loop accessing an array – named `arrayb`) and two memory assignment errors (both attempt to create a pointer in the mission memory pointing to an object in the private memory – named `mema1` and `mema2`). The patches for the `sim` version include, in addition to these, also a race condition (unprotected access to a buffer of radar frames that can make the detector thread read a corrupted or uninitialized radar frame – `race`). The patches for PapaBench include two memory assignment errors (named `mema3` and `mema4`), both of which attempt to create a pointer in the mission memory pointing to an object in the private memory.

We have verified that RSJ can check all our benchmarks both with and without the injected errors, that it finds the errors we have injected, and that it does not find any other (spurious) errors.

The checking time, memory usage, and number of visited and unique states are shown in Table 4. If an error is found, the testing is terminated. Executions that discovered an error are in the lower part of the table. Note that `cdx-l0-sim-race` is in the upper part of the table, because the race condition has not been detected. This is the desired behavior, as there cannot be a race condition in a level 0 program. Still, the race detector of JPF only detects suspected races, false alarms are thus possible in other examples. The reported amount of used memory is the maximum occupied memory as reported by the host virtual machine. It depends on the GC algorithm and the maximum heap size, and therefore only provides an approximate measure of the memory requirements of checking the particular program.

In most of the experiments, the number of visited and unique states was the same, and thus RSJ did not have to backtrack. This also means that most of the errors could have been discovered by non-exhaustive testing. The race condition is an exception, a specialized tool would still be needed to detect a potential race. The lack of backtracking also means that the program states are saved unnecessarily. This, however, is internal to JPF core, which requires a state to be saved at each thread switch, even if the switch is deterministic. This unnecessary saving of program states is most likely the cause why checking of PapaBench without errors needs so much memory.

The expected observations are that level 1 experiments take (almost always) as much time to check as level 0. Similarly, `sim` versions of `cdx` take longer to check than `nosim` versions.

4.4 Validation of a Timing Model

RSJ uses a timing model of the JOP processor, which provides lower and upper bounds for execution time on a given execution path (having all bytecode instructions in sequence). The timing simulator that implements this model in RSJ thus needs to support backtracking as RSJ uses backtracking while searching for errors. RSJ can also report execution time bounds to the tester in the form of logs. It can measure execution time of a complete plain Java program and computation times of individual releases of a periodic event handler (duration of `handleAsyncEvent`).

To validate that the timing model is implemented correctly in JPF, we run three plain Java benchmarks from the JBE (JavaBenchEmbedded) [22, 25] suite. The `kfl` and `lift` benchmarks are adaptations of two real-world applications, which are described in [23]. The `udpip` benchmark explores a tiny TCP/IP stack, which itself is also in industrial use. All applications are organized as single periodic tasks (similar to a cyclic executive). For benchmarking purpose we execute this periodic task 10000 times without waiting for the next period. We run the benchmarks on the 100 Mhz hardware (FPGA-based) implementation of JOP. We compare the measured time with the timing estimates given by RSJ. The results are summarized in Table 3. It can be seen that the best-case and worst-case execution times are quite

Benchmark	Measured Time	Estimated Time	
		Min.	Max.
`kfl`	483 ms	466 ms	577 ms
`lift`	486 ms	484 ms	514 ms
`udpip`	1094 ms	1042 ms	1194 ms

Table 3: Timing measured on JOP and estimated by R_{SJ}.

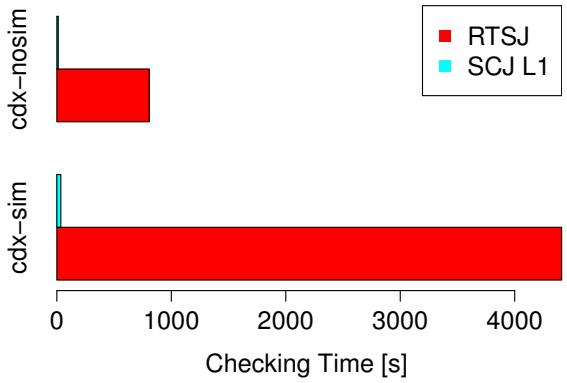

Figure 2: Checking time with RTSJ and SCJ L1, when no error is found.

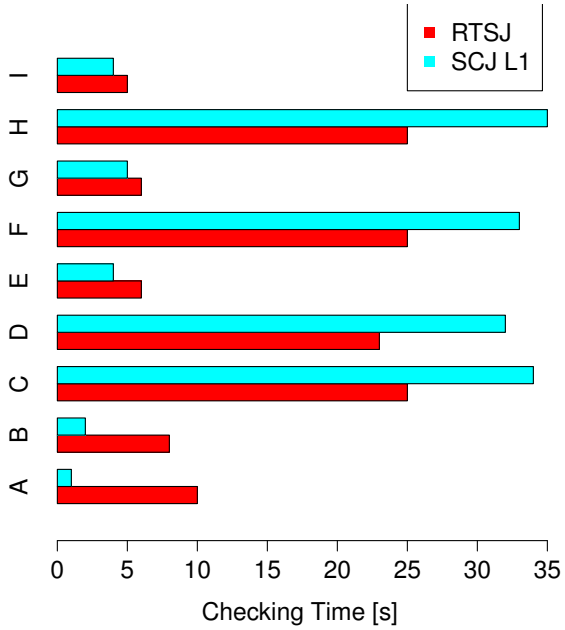

Figure 3: Checking time with RTSJ and SCJ L1, when an error is found. Letters stand for: `cdx-nosim-arrayb` (A), `cdx-sim-arrayb` (B), `cdx-nosim-mema1` (C), `cdx-sim-mema1` (D), `cdx-nosim-mema2` (E), `cdx-sim-mema2` (F), `cdx-sim-race` (G), `papabench-mema3` (H), and `papabench-mema4` (I).

close and the actual measurement in hardware is within this interval. Those results give us confidence that the timing model of JOP is sound. Note, that the measured maximum execution time and the maximum time reported by R_{SJ} are not the real WCET of the applications. It is the execution time of a run with a dedicated set of input data. Only static WCET analysis can give a sound WCET estimation for all possible input data [26].

4.5 Comparison with a Platform Independent Version

The scheduling algorithm we present in this paper takes advantage of the knowledge of the target platform in the form of a timing model. In our earlier work [19], we proposed and implemented a general algorithm that is platform independent. The general algorithm still generates sequences of release events such that they are consistent with periods and priorities of the periodic threads, but it makes no assumptions about how long code executes, which leads to testing of even unrealistic schedulings (e.g., that depend on thousands of bytecode instructions taking less time than a single bytecode instruction). Also, the general algorithm cannot estimate computation times or detect deadline misses. We compare the performance of R_{SJ} to the general tool R_J we presented in [19].

The results are presented in Table 5 and the checking times are also shown in Figure 2 (for no errors found) and Figure 3 (an error is found). The table shows that R_{SJ} needs less states for checking than R_J. In the cases errors are found, R_{SJ} is often slower even though it uses less states, which can be explained by internals of the implementation. R_{SJ} has to check if rescheduling is needed after every single bytecode instruction. Often, rescheduling is not needed, so no new state is created, but this decision has a measurable overhead. In the case no errors are found, and thus the whole state space is traversed, R_{SJ} is much faster than R_J. As can be seen in the table, R_J has to backtrack in these cases (the number of visited states is larger than the number of the unique states). This is because of the general algorithm that assumed no execution time model of the target platform. The timing model of JOP allowed R_{SJ} to avoid backtracking almost entirely in these experiments. The reduction of states is also significant: `cdx-rtsj-sim` used nearly 13G of memory, while `cdx-l1-sim` less than 1G. The checking of PapaBench without errors by R_J has run out of memory (20G heap limit). R_{SJ} can check PapaBench at SCJ level 1 in 30 minutes and 15G of memory.

4.6 Complexity of Used Benchmarks

We measure the complexity of the used benchmarks with the Chidamber and Kemerer (CK) object-oriented programming metrics [6], with the ckjm tool [27]. We calculate the metrics from every class the application loads, including standard Java libraries (but excluding real-time API that differs for different variants of the benchmarks).

Weighted methods per class (**WMC**) is the number of methods in a class. Depth of inheritance tree (**DIT**) is the number of ancestor classes of a class. Number of children (**NOC**) is the number of direct subclasses of a class. Coupling between object classes (**CBO**) is the number of classes coupled to a class. Two classes are coupled if one of them uses the methods or fields of the other class. This usage includes inheritance, method arguments, method types, and exceptions. Response for a class (**RFC**) is the number of

Benchmark	Checking Time	Memory Used	Unique States	Visited States
cdx-l0-nosim	0:00:08 s	486 M	16	16
cdx-l1-nosim	0:00:12 s	486 M	16	16
cdx-l0-sim	0:00:34 s	584 M	28	28
cdx-l1-sim	0:00:35 s	710 M	38	40
cdx-l0-sim-race	0:00:35 s	710 M	28	28
papabench-l0	0:15:28 s	13688 M	266431	266528
papabench-l1	0:30:42 s	14521 M	339254	339254
cdx-l0-nosim-arrayb	0:00:04 s	485 M	5	5
cdx-l1-nosim-arrayb	0:00:04 s	485 M	5	5
cdx-l0-sim-arrayb	0:00:33 s	675 M	10	10
cdx-l1-sim-arrayb	0:00:35 s	666 M	13	13
cdx-l0-nosim-mema1	0:00:04 s	485 M	5	5
cdx-l1-nosim-mema1	0:00:05 s	485 M	5	5
cdx-l0-sim-mema1	0:00:33 s	686 M	10	10
cdx-l1-sim-mema1	0:00:33 s	683 M	13	13
cdx-l0-nosim-mema2	0:00:04 s	485 M	5	5
cdx-l1-nosim-mema2	0:00:04 s	485 M	5	5
cdx-l0-sim-mema2	0:00:32 s	696 M	10	10
cdx-l1-sim-mema2	0:00:32 s	675 M	13	13
papabench-l0-mema3	0:00:02 s	482 M	39	39
papabench-l1-mema3	0:00:02 s	482 M	39	39
papabench-l0-mema4	0:00:02 s	482 M	41	41
papabench-l1-mema4	0:00:01 s	482 M	41	41
cdx-l1-sim-race	0:00:34 s	683 M	13	13

Table 4: Performance results of exhaustive testing of CDx and PapaBench.

	Benchmark	Checking Time	Memory Used	Unique States	Visited States
	cdx-l1-nosim	0:00:12 s	486 M	16	16
	cdx-rtsj-nosim	0:13:27 s	482 M	2096	2129
	cdx-l1-sim	0:00:35 s	710 M	38	40
	cdx-rtsj-sim	1:13:30 s	12819 M	72124	207398
A	cdx-l1-nosim-arrayb	0:00:04 s	485 M	5	5
A	cdx-rtsj-nosim-arrayb	0:00:05 s	482 M	14	14
B	cdx-l1-sim-arrayb	0:00:35 s	666 M	13	13
B	cdx-rtsj-sim-arrayb	0:00:25 s	482 M	57	60
C	cdx-l1-nosim-mema1	0:00:05 s	485 M	5	5
C	cdx-rtsj-nosim-mema1	0:00:06 s	482 M	12	12
D	cdx-l1-sim-mema1	0:00:33 s	683 M	13	13
D	cdx-rtsj-sim-mema1	0:00:25 s	482 M	57	60
E	cdx-l1-nosim-mema2	0:00:04 s	485 M	5	5
E	cdx-rtsj-nosim-mema2	0:00:06 s	482 M	14	14
F	cdx-l1-sim-mema2	0:00:32 s	675 M	13	13
F	cdx-rtsj-sim-mema2	0:00:23 s	482 M	57	60
G	cdx-l1-sim-race	0:00:34 s	683 M	13	13
G	cdx-rtsj-sim-race	0:00:25 s	482 M	56	59
H	papabench-l1-mema3	0:00:02 s	482 M	39	39
H	papabench-rtsj-mema3	0:00:08 s	655 M	55	55
I	papabench-l1-mema4	0:00:01 s	482 M	41	41
I	papabench-rtsj-mema4	0:00:10 s	655 M	65	65

Table 5: Performance of exhaustive testing in R_{SJ} (SCJ, JOP CPU) and R_J (RTSJ, CPU indep.).

	WMC			DIT			NOC			CBO		
	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum
cdx-sim	7	95	1254	1	5	164	0	4	29	6	62	881
cdx-nosim	7	95	537	1	4	58	0	3	9	5	20	292
papabench	3	34	684	1	3	137	0	14	31	7	39	934
kfl	6	21	73	1	2	10	0	1	1	2	5	25
udpip	5.5	10	63	1	2	16	0	2	4	4	11	60
lift	5	8	30	1	2	9	0	1	2	3	5	19

	RFC			LCOM			LOC		
	Med	Max	Sum	Med	Max	Sum	Med	Max	Sum
cdx-sim	15	117	2531	2	4465	11002	114.5	6023	36699
cdx-nosim	15	117	988	3	4465	6205	124	4151	13033
papabench	8.5	56	1367	1	465	3429	32	520	7402
kfl	9	41	124	10	90	200	67	506	1624
udpip	8.5	26	119	3	33	113	39.5	462	1523
lift	6	12	48	1	10	29	45	661	990

Table 6: Complexity of used benchmarks.

methods of the class plus the number of directly called external methods. Lack of cohesion (**LCOM**) is the number of methods of a class that do not share instance variables, which is calculated as $|P| - |Q|$, where P is the set of all method pairs that do not share any variable, and Q is the set of all pairs that do. Finally, lines of code (**LOC**) is the normalized number of lines of code of a class (supported by ckjm tool, but not an original CK metric). **LOC** is the sum of the number of methods, number of fields, and number of byte-code instructions of methods. The metrics are summarized in Table 6. With respect to scheduling, PapaBench is the most complex benchmark with 14 periodic event handlers, followed by `cdx-sim` with 2 periodic event handlers. All other benchmarks have only a single task.

5. RELATED WORK

Model checking of real-time Java programs using JPF has first been studied in [15]. The primary focus of the work was to verify that all threads in an RTSJ applications meet their deadlines. The basic assumption made was that execution time of a bytecode instruction is independent of the context: the timing model uses a table that maps an instruction type to the assumed duration. The timing model does not seem to be motivated by existing hardware. The simplicity of the timing model makes it easy to design the scheduling algorithm as a discrete event simulation. The work only includes a very small subset of RTSJ, particularly it does not include scoped and immortal memory areas that we support. The key advantage of our approach is, however, a more general timing model that allows execution context to be incorporated into the timings: our timing model allows for instance to incorporate caches, and is motivated by and validated against a real hardware implementation.

There are several tools for verification of real-time systems based on abstract models of these systems, such as [14, 28, 30]. RSJ complements these tools as it can work directly with code, thus potentially finding errors not visible at the model level. On the other hand, due to state explosion RSJ can only fully traverse relatively small programs. Indeed, RSJ also could be used with models, if SCJ programs passed to it were indeed models of real systems.

Some work has been done on preventing memory assignment errors at compile time, with the help of a special type system and code annotations that have to be added by the programmer [5, 31]. The advantage of these approaches is that they can provably prevent the memory access errors, but for the additional burden of manual code annotations. The limitation is also that only local variables are supported [5] or certain programming restrictions are imposed [31]. RSJ can search for memory assignment errors in unannotated programs with no programming restrictions in addition to those imposed by Java and SCJ.

Using an execution time model of a target platform to extract a performance metric on a host with JPF is similar to cross-profiling [3]. With cross-profiling the program under test is instrumented at basic block boundaries to collect the performance metric and executed on a standard (JIT compiling) JVM. With this approach only the average case execution time is considered, whereas RSJ considers best-case and worst-case timings for given input data.

6. CONCLUSION

We provide RSJ, a testing tool that can exhaustively search for errors in SCJ programs (SCJ levels 0 and 1 without aperiodic event handlers), backtracking over different scheduling orders possible on a real system. The tool can thus discover some corner case errors that are hard to find by regular testing. The tool is an extension of Java PathFinder, an execution based model checker for Java. Consequently, various kinds of run-time analyses are possible. The tool can currently calculate execution time bounds based on (fixed) inputs of the tests. Also, it can detect potential race conditions. It also provides a base for further kinds of analyses, such as establishing memory bounds for scoped memories or the stack memory.

To test RSJ, we provide an SCJ version of the PapaBench benchmark, which is a non-trivial application benchmark that consists of hard real-time code that has, in its C original, flown real UAVs. To our knowledge, the SCJ version of PapaBench is the publicly available real-time Java benchmark with most complex real-time application logic to date.

RSJ uses our novel scheduling algorithm for an execution-based model checker that can verify real-time programs. Our algorithm uses a realistic execution time model of JOP, a hardware JVM implementation. We show that the use of a platform-specific model can drastically speed-up the exhaustive testing when compared to a platform-independent algorithm by essentially eliminating backtracking. This is possible even with a timing model that defensively estimates timings as intervals, allowing for instance modeling of caches. This is an interesting result in the field of model checking, where the general understanding is that concurrency always leads to state space explosion. We show that it needs not to be the case for real-time programs.

At the same time, it appears that new and significant optimizations are needed in model checkers for real-time programs. As the scheduling is more deterministic than for non-realtime, it is a crude waste of time and memory to store a new state of the program at every scheduling-relevant instruction, including accesses to shared fields. States should only be created at choice points that actually include at least two possible schedules. Note that in non-realtime concurrent programs, this would hardly have been an issue, as all runnable threads compete for the processor at all times. Also, the notion of time in the model checker, which has to be added for checking of real-time programs, requires a new view on state matching. Making the current time (or time estimate) part of the program state would hardly ever render two states equal. Excluding it completely, as done by Java PathFinder, is incorrect as it only allows to traverse a part of program state space. We discovered this problem in this work, but we have not solved it. We just disabled state matching in RSJ to get correct results.

The most significant limitation of RSJ with respect to support of SCJ API is the lack of support for aperiodic event handlers. While this could in theory be fixed in the present explicit-state solution, such a fix would inevitably lead to state space explosion. A good solution which we do not provide in this work could be based on symmetry reductions or abstractions computed using static analysis.

Acknowledgments

This work was partially supported by the Ministry of Education of the Czech Republic grant MSM0021620838.

7. REFERENCES

[1] Java Path Finder. http://babelfish.arc.nasa.gov/trac/jpf/, 2010.

[2] Paparazzi: The free autopilot. http://paparazzi.enac.fr/, 2010.

[3] Walter Binder, Martin Schoeberl, Philippe Moret, and Alex Villazon. Cross-profiling for Java processors. *Soft. Pract. Exp.*, 39/18, 2009.

[4] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[5] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time java. *SIGPLAN Not.*, 38(5), 2003.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Soft. Eng.*, 20(6), 1994.

[7] EUROCAE ED-12B software considerations in airborne systems and equipment certification, 1992.

[8] Patrice Godefroid. Model checking for programming languages usingVeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1997.

[9] Trevor Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.

[10] Thomas Henties, James Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *Certification of Safety-Critical Software Controlled Systems (SafeCert)*, 2009.

[11] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[12] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. CDx: A family of real-time Java benchmarks. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2009.

[13] Tomas Kalibera, Pavel Parizek, Ghaith Haddad, Gary T. Leavens, and Jan Vitek. Challenge benchmarks for verification of real-time programs. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification (PLPV)*, 2010.

[14] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2), 1997.

[15] Gary Lindstrom, Peter C. Mehlitz, and Willem Visser. Model checking real time Java using Java PathFinder. In *Proceedings of Automated Technology for Verification and Analysis, Third International Symposium (ATVA)*, 2005.

[16] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: a pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI), 2002.

[17] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6), 2007.

[18] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *Proceedings of 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

[19] Pavel Parizek, Tomas Kalibera, and Jan Vitek. Model checking real-time Java. Technical Report 1, Dept. of Distributed and Dependable System, Charles University, http: //d3s.mff.cuni.cz/publications/rtJavaChecking.pdf, 2010.

[20] Software considerations in airborne systems and equipment certification, 1992.

[21] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *Proceedings of the International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, 2004.

[22] Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, Vienna, Austria, 2005.

[23] Martin Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, 2008.

[24] Martin Schoeberl. A Java processor architecture for embedded real-time systems. *J. Sys. Arch.*, 54/1–2, 2008.

[25] Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, 2010.

[26] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a Java processor. *Soft. Pract. Exp.*, 40/6, 2010.

[27] D. D. Spinellis. ckjm - A Tool for Calculating Chidamber and Kemerer Java Metrics. http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/, 2009.

[28] Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for real time. In *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 1996.

[29] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2), 2003.

[30] Sergio Yovine. Kronos: A verification tool for real-time systems. *STTT*, 1(1-2):123–133, 1997.

[31] Tian Zhao, James Noble, and Jan Vitek. Scoped types for real-time java. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, 2004.